חיפוש בבינה מלאכותית

(מבוסס על הרצאות של דר' אריאל פלנר וספרו של פרופ' ריצ'רד א. קורף מ-UCLA)

פרק 1: בעיות ומרחבי בעיות

סוגי בעיות:

1. single agent path finding: להגיע מהמיקום ההתחלתי למטרה – מעניין אותנו אופן ההגעה (הצעדים) יותר מאשר הפתרון (לדוגמה: קובייה הונגרית, tile puzzle, TSP).
2. Two players game: שח, דמקה, שש-בש, אותלו.
3. Constraint satisfaction: סוכן אחד מבצע פעולות. מעניין אותנו הפתרון עצמו ולא אופן ההגעה אליו (לדוגמה: בעיית 8 המלכות, number partioning: חלוקת מספרים ל-2 תתי-קבוצות כך שסכום כל קבוצה יהיה כמה שיותר קרוב).

ניתן להמיר בין בעיות בייצוגים שונים. לדוגמה, ניתן להמיר בעיית TSP לייצוג CSP ע"י הגדרת האילוצים שכל פתרון חייב לעמוד בהם, כולל קריטריון האופטימיזציה.

הגדרה:
מרחב הבעיה מוגדר על-ידי **מצבים** (הקונפיגורציה של הבעיה) ו**אופרטורים** (פונקציה שממפה מצב למצב אחר).
מופע הבעיה מוגדר ע"י מרחב הבעיה, מצב התחלתי ואוסף של מצבי מטרה (goal states). המצב הסופי יכול להיות מפורש (explicit - המצב עצמו) או לא מפורש (implicit - הגעה למצב מטרה נבדקת ע"י בדיקות חוקיות למיניהן).
לדוגמה, ב-CSP, כיוון שתמיד מעניין אותנו הפתרון עצמו לא ייתכן שהמצב הסופי יינתן בצורה מפורשת.

אופן ייצוג הבעייה יכול להשפיע רבות על יעילות הפתרון. כלל אצבע: **ככל שייצוג הבעיה קטן יותר** (פחות מצבים לחפש) – **ככה הוא יותר יעיל**.

ב**גרף מרחב הבעיה** הקודקודים מייצגים מצבים והקשתות את האופרטורים.

כשיש יותר מ-2 דרכים להגיע למצב מסוים ניתן לזהות זאת רק ע"י שמירת כל המצבים שנוצרו בעבר וביצוע השוואות. אולם בשל הפשטות ורצון לחיסכון בזיכרון רוב האלג' לא עושים זאת.

Branching factor של קודקוד – יחס ההסתעפות = מס' הבנים של קודקוד (לא כולל הורים במידה והאופרטור הפיך).
Branching factor של מרחב בעיה – המספר הממוצע של הבנים של הקודקודים השונים.
Solution depth (עומק הפתרון) – אורך המסלול הקצר ביותר מהקודקוד ההתחלתי לקודקוד המטרה.

צמצום קודקודים כפולים: לא נפעיל אופרטור וישר אחריו נפעיל את האופרטור ההופכי, שכן מסלול אופטימלי לא יכלול נתיב כזה. לכן, לא נגדיר אבא של קודקוד כאחד מילדיו ← הקטנת יחס ההסתעפות בערך ב-1.

סוגי מרחבי בעיות:

1. **State Space**: המצבים מייצגים את מצבי הבעיה והאופרטורים את הפעולות בעולם.
   סוגי חיפושים:
   - **forward search** – מהשורש, שמייצג את המצב ההתחלתי, ועד המטרה.
   - **Backward search** – מהשורש, שמייצג את המטרה, ועד המצב ההתחלתי.

   על מנת לאפשר backward search חובה שמצב המטרה יוגדר בצורה מפורשת. **אין חובה** שהאופרטורים יהיו הפיכים, צריך לדעת רק כיצד ניתן להגיע למצב בצורה הפוכה.

דוגמאות: קובייה הונגרית, tile puzzle.

2. **problem reduction space**: הקודקודים מייצגים בעיות שיש לפתור והקשתות מייצגת פירוק הבעיות לתתי בעיות (לדוגמה: מגדלי הנוי).

3. **and/or graphs**:
and graph – כל הקודקודים מייצגים AND, וכדי לפתור את הבעיה יש לפתור את כל הבעיות המיוצגות ע"י כל הילדים (לדוגמה: מגדלי הנוי). הפתרון הוא **כל** הגרף.
or graph – כל הקודקודים מייצגים OR, וכדי לפתור את הבעיה יש לפתור את הבעיה המיוצגת ע"י אחד מהקודקודים (לדוגמה: 8 puzzle). הפתרון הוא **נתיב** מהשורש לקודקוד המטרה.
and/or graph – כולל גם קודקודי AND וגם קודקודי OR. לדוגמה: כשלא ניתן לחזות את השלכות פעולה מסוימת מראש (לדוגמה: counterfeit coin problem, 5 אבנים). הפתרון הוא תת-גרף כך ש:
א) הוא כולל את השורש.
ב) לכל קודקוד AND שנכלל בפתרון כל הילדים נכללים גם כן.
ג) לכל קודקוד OR שנכלל בפתרון רק קודקוד ילד אחד נכלל.

יש 3 מידות לבחינת יעילות האלגוריתם:
1. **איכות הפתרון** (עד כמה אופטימלי).
2. **סיבוכיות זמן הריצה**.
3. **סיבוכיות הזיכרון**.

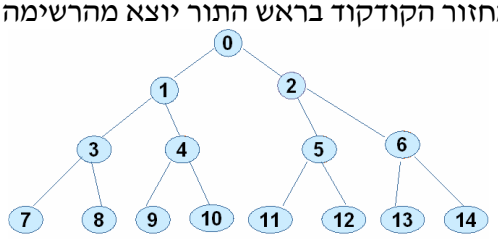
פרק 2: Brute Force Search

אלגוריתמים אלו לא משתמשים ב-domain specific knowledge. האלגוריתמים משתמשים רק במרחב הבעיה ובתיאור מצב המטרה כדי למצוא פתרון.

הם דורשים: תיאור מצב, אופרטורים, מצב התחלתי, תיאור מצב המטרה.
לכל הקודקודים מוצמדת עלות.
יצירת קודקוד = יצירת מבנה הנתונים המתאים לקודקוד.
הרחבת קודקוד = יצירת מבני הנתונים המתאימים לכל הבנים של הקודקוד.

Breadth First Search (BFS):

הרחבת הקודקודים בהתאם למרחקם מהשורש (לפי רמת הקודקוד – level). האלגוריתם יוצר רמה אחת של העץ בכל פעם עד למציאת הפתרון.
ממומש ע"י FIFO – בכל מחזור הקודקוד בראש התור יוצא מהרשימה ובניו נכנסים **לסוף** התור.



The numbers represent the order generated by BFS

איכות הפתרון: אם קיים קודקוד מטרה, BFS ימצא אותו, שכן BFS יוצר את כל הקודקודים בכל רמה לפני שממשיך לרמה הבאה.
2 אפשרויות לייצוג הפתרון:
א. לכל קודקוד נשמר מהלך הצעדים שבוצעו עד אליו.
ב. לכל קודקוד נשמר מצביע לאבא – יעיל יותר מבחינת זיכרון.

סיבוכיות זמן: במקרה הגרוע ביותר יש לייצר את כל הקודקודים בגובה d. אם נסמן ב-N(b,d) את מספר הקודקודים שנוצרו (בהנחה שיצירת כל קודקוד לוקחת זמן יחידה), b מייצג את ה-branching factor ו-d את גובה הפתרון אזי:

$$N(b,d) = 1 + b + b^2 + \ldots + b^d = \frac{b^{d+1}-1}{b-1} \approx b^d\left(\frac{b}{b-1}\right) = \mathbf{O(b^d)}$$

סיבוכיות זיכרון: כדי לדווח על הפתרון צריך לאחסן את כל הקודקודים שנוצרו ← $\mathbf{O(b^d)}$.

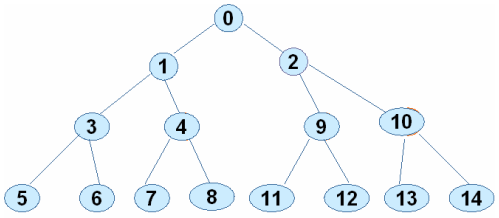
BFS וכל אלגוריתם שחייב לאחסן את כל הקודקודים הינו **חסום מבחינת הזיכרון** וינצל את זיכרון המחשב עד סופו תוך דקות.

Depth First Search (DFS):

יצירת הילד הבא של הקודקוד העמוק ביותר שעוד לא הורחב סופית.

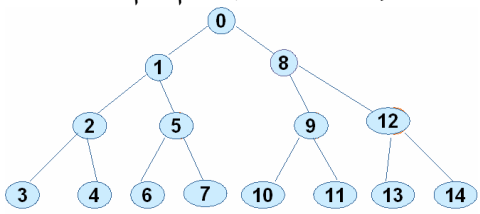
2 דרכי מימוש:

א. **LIFO**: בכל מחזור, הקודקוד שבראש המחסנית מוצא ומורחב, וילדיו מוכנסים **לראש** המחסנית.



The numbers represent the order generated by DFS

ב. **רקורסיה**: מקבלת קודקוד ומבצעת DFS מתחת לקודקוד.



The numbers represent the order generated by DFS

סיבוכיות זיכרון: היצירה מאחסנת O(d) קודקודים (האלגוריתם צריך לאחסן רק קודקודים שנמצאים בנתיב מהשורש לקודקוד הנוכחי).
ההרחבה מאחסנת O(bd) קודקודים.

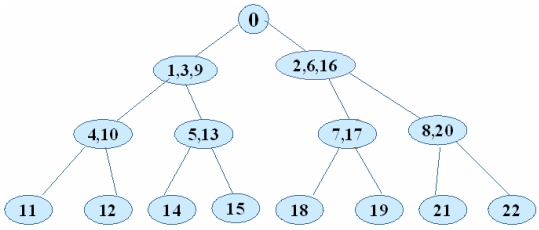
סיבוכיות זמן: DFS מייצר אותה קבוצת קודקודים כמו BFS (רק בסדר שונה) ← $O(b^d)$.
עבור עצים אינסופיים DFS **לא יסתיים** (לדוגמה: 8 puzzle).
כדי להתמודד עם עצים אינסופיים מוסיפים **cutoff depth** כך שהחיפוש מבוצע עד גובה זה:
אם ה-d < cutoff depth ← האלגוריתם לא ימצא פתרון.
אם ה-d > cutoff depth ← האלגוריתם ימצא פתרון אבל סיבוכיות הזמן גבוהה מזו של BFS.

DFS הינו יותר **חסום זמן** מאשר חסום זיכרון.

איכות הפתרון: הפתרון הראשון ש-DFS מוצא אינו בהכרח האופטימלי (יוחזר הפתרון ה"שמאלי" ביותר).

Depth First Iterative Deepening (DFID):

שילוב של BFS ו-DFS: מבוצע DFS ברמה 1, אח"כ מבוצע DFS לרמה 2 וכן הלאה, עד שנמצא פתרון.



The numbers represent the order generated by DFID

איכות הפתרון: כיוון שלא נוצרים קודקודים עד שהרמה הנמוכה ביותר נוצרה ← הפתרון הראשון שנמצא הינו בנתיב הקצר ביותר.

סיבוכיות זיכרון: כמו DFS – בכל רגע נתון הוא שומר רק מחסנית של קודקודים – O(d).

סיבוכיות זמן: מספר הקודקודים שנוצרים ע"י DFID הוא: $b^d(\frac{b}{b-1})^2$ ← $O(b^d)$.

רוב העבודה של האלגוריתם נעשית באיטרציה האחרונה.

לכן, היחס בין מס' הקודקודים שנוצרים ע"י DFID ואלו הנוצרים ע"י BFS הוא $(\frac{b}{b-1})$.

משפט: DFID הינו אלגוריתם אופטימלי אסימפטוטית במובן של זמן וזיכרון בהשוואה לשאר אלגוריתמי brute force search עם משקל זהה לקשתות [הוכחה ע"י השוואת סיבוכיות הזמן (מניחים בשלילה שקיים אלגוריתם יותר מהיר ומראים שבגרף אחר שבו יש רק קודקוד מטרה אחד האלגוריתם לא יזהה אותו), סיבוכיות הזיכרון ואיכות הפתרון].

כל אלגוריתם שלוקח f(n) זמן חייב להשתמש לפחות ב-logf(n) זיכרון. לכן, אם האלגוריתם לוקח $O(b^d)$ זמן הוא חייב להשתמש לפחות ב-O(d) זיכרון.

<u>גרף עם מעגלים:</u>

בגרף עם מעגלים BFS יותר יעיל מ-DFS ו-DFID שכן הוא יכול לזהות את כל הקודקודים הכפולים.

סיבוכיות ה-BFS תלויה במספר **הקודקודים** בעומק מסוים.

סיבוכיות ה-DFS תלויה במספר **הנתיבים** בעומק מסוים.

בגרף בו יש מספר רב של מעגלים קצרים, BFS עדיף על DFS בהנחה שיש מספיק זיכרון.

<u>לסיכום:</u>

- BFS, DFS, DFID יוצרים אסימפטוטית אותו מספר קודקודים.
- DFS, DFID יעילים יותר מ-BFS.
- הזמן שלוקח ליצור קודקוד הינו פרופורציוני לגודל ייצוג המצב.
- יתרון מימוש DFS בצורה רקורסיבית בא לידי ביטוי ככל שייצוג המצבים גדול יותר.

<u>Bidirectional Search:</u>

חיפוש סימולטנית קדימה מהמצב ההתחלתי ואחורה ממצב המטרה, עד אשר נפגשים במצב זהה.

חיפוש זה מבטיח מציאת נתיב קצר ביותר מהמצב ההתחלתי למטרה, אם קיים.

בהנחה שיש פתרון בעומק d וששני החיפושים הם מסוג BFS אזי נגיע למצב t שמרחקו k מההתחלה ו-$(d-k)$ מהמטרה.

סיבוכיות זמן: אם שני החיפושים נפגשים באמצע, אזי כל חיפוש הגיע לעומק $\frac{d}{2}$ ← $O(b^{d/2})$, אבל, צריך בנוסף להשוות כל קודקוד חדש עם כל הקודקודים מהצד השני (כדי לבדוק זהות) וגם זה $O(b^{d/2})$, ולכן סה"כ הסיבוכיות היא **$O(b^d)$**. ניתן לשפר סיבוכיות זאת ע"י שימוש ב- hash table להשוואות, ואז כל השוואה נעשית בזמן קבוע ולכן הסיבוכיות תהיה $O(b^{d/2})$.

סיבוכיות מקום אופטימלית תושג ע"י מימוש אלגוריתם אחד כ-BFS והשני ע"י DFS או DFID. כיוון שלפחות אחד הנתיבים חייב להיות מאוחסן בזיכרון, סיבוכיות המקום נשלטת ע"י סיבוכיות המקום של BFS ולכן היא $O(b^{d/2})$.

בגרף סופי חיפוש דו-כיווני עשוי להיות **לא יותר יעיל** מחיפוש חד-כיווני במקרה הגרוע ביותר.

bidirectional search הינו **חסום מקום** אבל יותר יעיל מבחינת זמן מ-unidirectional search.

פרק 3: Best First Search

נניח שלכל קשת יש משקל שרירותי, לאו-דווקא זהה. best first search הינה מחלקה של אלגוריתמי חיפוש אשר משתמשים בפונקציית משקל (cost). האלגוריתמים עצמם שונים בהתאם לפונקציית המשקל שהם מיישמים.
אלגוריתמי ה-best first search דורשים מקום אקספוננציאלי.

האלגוריתמים הנ"ל משתמשים ב-2 רשימות של קודקודים:
א. **closed list** – רשימת כל הקודקודים שהורחבו לגמרי.
ב. **open list** – רשימת כל הקודקודים שנוצרו אבל טרם הורחבו. רשימה זו ממומשת כ**תור עדיפויות**.

בתחילת האלגוריתמים open = {root}, closed = {}.

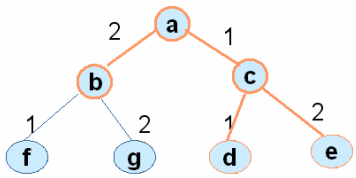
בכל איטרציה הקודקוד בעל העלות הנמוכה ביותר שנמצא ב-open list מורחב, מועבר ל-closed list וילדיו מועברים ל-open list.
האלגוריתם מסתיים כשנבחר קודקוד מטרה להרחבה או כשאין יותר קודקודים ב-open list.

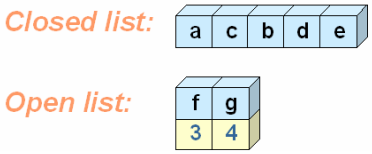
BFS הוא דוגמה לאלגוריתם מסוג best first עם פונקציית עלות שהיא **עומק הקודקוד** מהשורש.
DFS אינו נחשב כאלגוריתם best first שכן כדי לרוץ בזיכרון לינאי הוא אינו שומר open/closed lists.

קודקוד המטרה מוחזר רק ברגע שהוא נבחר **להרחבה** ולא ברגע שהוא מתווסף ל-open list.

Uniform Cost Search (UCS):

g(n) – סכום עלות הקשתות מהשורש ועד קודקוד n.
אם g(n) היא פונקציית העלות הכללית שלנו אזי האלגוריתם הוא מסוג UCS או Dijkstra's single source shortest path.





UCS נחשב כ-brute force search כי הוא אינו משתמש בפונקציה היוריסטית.

UCS ימצא את קודקוד המטרה או ידווח שאין פתרון אם:
א. מרחב הבעיה סופי.
ב. יש מסלול למטרה שהוא באורך סופי ומשקל סופי.
ג. אין מסלולים אינסופיים עם משקל סופי.

נניח שלכל הקודקודים יש משקל מינימלי חיובי e לפתור בעיה של לולאה אינסופית של קשתות בעלות משקל 0, אזי UCS יגיע למטרה (בהנחה שקיימת) עם משקל סופי.

איכות הפתרון – משפט: בגרף בו לכל הקשתות יש משקל מינימלי חיובי, וקיים מסלול סופי לקודקוד מטרה, UCS יחזיר מסלול בעל משקל מינימלי למטרה [הוכחה: א) להראות שאם ב-open list יש קודקוד במסלול אופטימלי לפני הרחבה, אזי יהיה קודקוד כזה גם אחרי הרחבה (אינדוקציה); ב) אם קיים מסלול למטרה אז האלגוריתם ימצא אותו בסוף; ג) בפעם הראשונה שקודקוד מטרה נבחר להרחבה האלגוריתם יסתיים ויחזיר את המסלול לקודקוד זה כפתרון].

סיבוכיות זמן: במקרה הגרוע ביותר לכל קודקוד יש משקל e, c הינו משקל הפתרון האופטימלי ולכן כאשר כל הקודקודים עם משקל קטן/שווה c נבחרו להרחבה, קודקוד המטרה חייב להיבחר גם כן. האורך המקסימלי של כל נתיב עד לנקודה זו לא יכול להיות יותר מ- c/e ולכן המספר הגרוע ביותר של קודקודים הוא $b^{c/e}$ ← $O(b^{c/e})$.

כיצד אם כן מוגדר שסיבוכיות האלגוריתם של Dijkstra הינה $O(n^2)$?
הסיבוכיות הינה $O(|E|) = O(n^2)$, כאשר n הינו מספר הקודקודים בגרף, ואילו אצלנו מספר הקודקודים מיוצג ע"י b ו-c.
בנוסף, ב-Dijkstra מניחים שכל קודקוד מחובר לקודקוד אחר ולכן הסיבוכיות הריבועית. ב-UCS מניחים branching factor קבוע b.
ב-Dijkstra ה-open list כולל את כל השכנים שטרם הורחבו של הקודקודים שכבר הורחבו.

סיבוכיות מקום: כמו בכל best first search, כל קודקוד שנוצר מאוחסן באחת הרשימות, ולכן סיבוכיות הזיכרון זהה לסיבוכיות הזמן - $O(b^{c/e})$.
לכן ה-UCS הינו **חסום מקום**.

<u>פונקציות היוריסטיות:</u>
הפונקציות היוריסטיות משמשות להובלת האלגוריתם לכיוון המטרה או לקיצוץ ענפים שאינם בנתיב של המטרה.

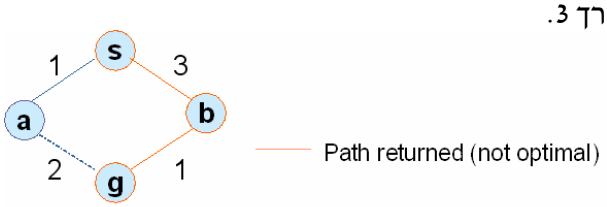
2 התכונות החשובות ביותר:
1) הפונקציה מייצגת בצורה טובה הערכה של העלות להגיע למטרה.
2) הפונקציה היא קלה יחסית לחישוב.

תכונה נוספת:
3) **admissibility** – הפונקציה היא תמיד **חסם תחתון** (**lower bound**) לפתרון עצמו.

<u>Pure Heuristic Search (PHS):</u>
$f(n) = h(n)$, כאשר f הינה פונקציית העלות (כמה עולה להגיע לקודקוד הבא) ו-h הינה הפונקציה היוריסטית אשר מציינת **כמה עוד** נשאר.
PHS ייצור בסופו של דבר את כל הגרף וימצא את קודקוד המטרה, אם קיים כזה.
אם הגרף אינסופי לא מובטח ש-PHS יסיים, אפילו אם קיים קודקוד מטרה.
אם PHS מסיים עם פתרון, **לא מובטח שהפתרון אופטימלי**.

PHS הינו אלגוריתם טוב למציאת פתרונות sub-optimal לבעיות קומבינטוריות.

בדוגמה הבאה המספרים מייצגים את $h()$, והמסלול שמוחזר הינו באורך 4 לעומת המסלול האופטימלי שהינו באורך 3.



כדי למצוא אלגוריתם אופטימלי צריך לקחת בחשבון את **העלות** להגיע לכל קודקוד ב-open list מהקודקוד ההתחלתי **וגם** את ההערכה היוריסטית **לכמה נותר** לקודקוד המטרה:

$f(n) = W_g g(n) + W_h h(n)$, כאשר $\frac{W_h}{W_g} = W$, $1 < W < \infty$.

[g = כמה הלכתי מההתחלה, h = כמה עוד נותר]

A*:

לוקחים בחשבון את העלות להגיע לקודקוד מהקודקוד ההתחלתי – $g(h)$, וגם את ההערכה היוריסטית להגיע מהקודקוד הנוכחי למטרה – $h(n)$: $f(n) = g(n) + h(n)$.

כמו שאר אלגוריתמי best-first search, A* מסתיים כשהוא בוחר קודקוד מטרה להרחבה, או כאשר אין יותר קודקודים ב-open list.

בגרף אינסופי הוא ימצא מסלול עם עלות סופית אם:

א. כל הקודקודים בעלי עלות סופית ובעלי ערך חיובי מינימלי.

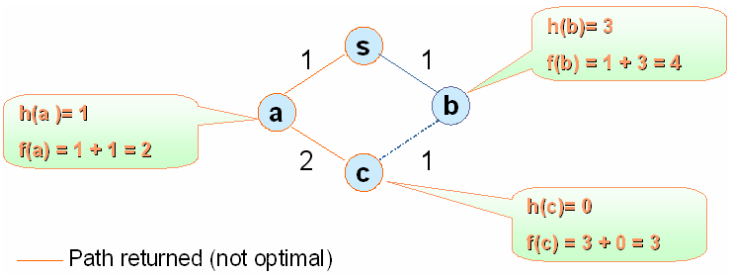
ב. כל ערכי היוריסטיקה סופיים ולא שליליים.

המטרה העיקרית של ה-closed list ב-A* היא לשפר את זמן הריצה בגרפים עם מעגלים, זאת ע"י זיהוי מצבים שהורחבו בעבר והימנעות מהרחבתם בשנית.

משפט: בגרף בו לכל הקודקודים יש משקל חיובי מינימלי, וערכי היוריסטיקה לא שליליים ואף פעם אינם overestimate של העלות האמיתית, וקיים מסלול עם עלות סופית למטרה, A* יחזיר מסלול אופטימלי למטרה [הוכחה: א) להראות שאם ב-open list יש קודקוד במסלול אופטימלי לפני הרחבה, אזי יהיה קודקוד כזה גם אחרי הרחבה; ב) אם קיים מסלול למטרה אז האלגוריתם ימצא אותו בסוף; ג) בפעם הראשונה שקודקוד מטרה נבחר להרחבה האלגוריתם יסתיים ויחזיר את המסלול לקודקוד זה כפתרון].

← כל אלגוריתם חיפוש אופטימלי אחר ירחיב לפחות את כל הקודקודים שהורחבו ע"י A*.

עבור פונקציה $h()$ שהיא overestimate, A* לא יחזיר בהכרח את הפתרון האופטימלי.
לדוגמה:



<u>תכונות שונות של פונקציות היוריסטיות:</u>

א. **admissible**: בהנחה ש-$h^*(n)$ הינה העלות האמיתית למטרה מקודקוד n, אזי $h(n)$ הינה admissible אם"ם לכל n: $h(n) \le h^*(n)$.

ב. **consistent**: אם $c(n,m)$ היא העלות למסלול הקצר ביותר מ-n ל-m, אזי $h(n)$ היא consistent אם לכל n: $h(n) \le c(n,m) + h(m)$.

ג. **monotonic**: אם $h(m)$ היא קונסיסטנטית אזי פונקצית העלות $f(n)$ היא מונוטונית לא-יורדת $f(n) \le f(n')$, כאשר n' הינו קודקוד ילד של n. בצורה דומה, אם f מונוטונית אזי h קונסיסטנטית.

$$h(n) \le c(n,n') + h(n')$$
$$g(n) + h(n) \le g(n) + c(n,n') + h(n')$$
$$g(n) + h(n) \le g(n') + h(n')$$
$$f(n) \le f(n')$$

בנוסף, אם הפונקציה היא קונסיסטנטית אזי היא בהכרח אדמיסבילית (אבל לא להיפך):
$$h(n) \le c(n,m) + h(m)$$
$$h(n) \le c(n,G) + h(G)$$
$$h(n) \le c(n,G) + 0$$
$$h(n) \le h^*(n)$$

בהינתן פונקציה שהיא אדמיסבלית אבל לא קונסיסטנטית ניתן לבנות פונקציה מונוטונית f שהיא אדמיסבלית: $f(n') := \max(f(n'), f(n))$.
במקרה זה, אם $h(n)$ אדמיסבילית אזי גם $f(n)$ אדמיסבילית שכן $f(n) = g(n) + h(n)$ הינו חסם תחתון על העלות הכוללת להגיע למטרה מהמצב ההתחלתי. לכן, העלות הכוללת להגיע למטרה מכל קודקוד בן חייבת להיות **לפחות** גדולה כמו העלות המינימלית דרך האבא.

סיבוכיות זמן:
סיבוכיות הזמן פרופורציונלית למספר הקודקודים שנוצרים או מורחבים. לכן, ה-branching factor הוא לכל היותר **קבוע**, וההערכה היוריסטית של קודקוד יכולה להתבצע בזמן קבוע.
את ה-open list וה-closed list ניתן לתחזק בזמן קבוע לכל הרחבת קודקוד:
closed list: ניתן להחזיק כ-hash table שכן צריך רק לבדוק קיום קודקוד.
open list: צריך להוסיף קודקוד ולהוציא קודקוד בזמן קבוע. זה ייקח זמן שהינו לוגריתמי בגודל הרשימה.
במקרה בו ערכי היוריסטיקה הינם מספרים שלמים או בעלי מספר מצומצם של ערכים שונים, ניתן לשמור את ה-open list כמערך של רשימות, כל רשימה מייצגת עלות שונה. מימוש זה מאפשר זמן קבוע להוצאה/הכנסה.
לכן, כדי למצוא את סיבוכיות הזמן צריך למצוא כמה קודקודים A* יוצר במהלך מציאת הפתרון.
הפתרון עצמו תלוי באיכות הפונקציה היוריסטית.

<u>המקרה הגרוע ביותר: $f(n) = g(n)$</u>
$h(n) = 0$ לכל קודקוד. h עדיין מהווה חסם תחתון לעלות. אלגוריתם זה שקול ל-UCS, ולכן סיבוכיות הזמן הינה $O(b^{c/e})$.

<u>המקרה הטוב ביותר: $f(n) = g(n) + h^*(n)$</u>
$h(n) = h^*(n)$ לכל קודקוד. סיבוכיות הזמן: $O(bd) = O(d)$.

<u>שבירת שוויון:</u>
נניח שכל הקודקודים בעומק d הינם קודקודי מטרה, וכל מסלול לקודקוד הוא אופטימלי. לכן, לכל קודקוד יהיה אותו משקל. אם שבירת השוויון נעשית לטובת קודקודים עם $g()$ מינימלי, נצטרך ליצור את כל העץ לפני שנגיע לקודקוד המטרה, ולכן סיבוכיות הזמן תהיה $O(b^d)$, למרות שיש לנו פונקציה היוריסטית מושלמת.

אם שבירת השוויון לקודקודים בעלי אותו ערך $f(n)$ תבוצע לפי ערך ה-$h(n)$ המינימלי (או $g(n)$ המקסימלי) נקבל סיבוכיות זמן טובה יותר. זה מבטיח שכל שוויון יישבר לטובת קודקוד מטרה (שכן $h(G) = 0$) וסיבוכיות הזמן של A* עם היוריסטיקה מושלמת תהיה O(d).

<u>אופן הרחבת הקודקודים:</u>
אם הפונקציה היוריסטית היא קונסיסטנטית אזי $f(n)$ היא מונוטונית לא יורדת לאורך כל מסלול מהשורש. לכן, כל הקודקודים עם העלות האופטימלית $f(n) < c$ יורחבו תמיד, אף קודקוד $f(n)$ $c <$ לא יורחב, וחלק מהקודקודים $f(n) = c$ יורחבו.
לכן, $f(n) < c$ הינו **תנאי מספיק** עבור A* להרחבת קודקוד n ו- $f(n) \le c$ הינו **תנאי הכרחי**.

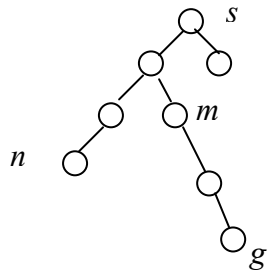
Weighted A* (WA*):

$f(n) = W_g g(n) + W_h h(n)$, כאשר $\frac{W_h}{W_g} = W$, $1 < W < \infty$.

כאשר $W = 1$ מקבלים את A*.
כאשר $W = \infty$ מקבלים את PHS.

Absolute Error:
נניח שמרחב הבעיה הוא עץ, ללא מעגלים. יש branching factor אחיד b (כלומר, לכל קודקוד יש b בנים), כל קשת או אופרטור הם בעלי עלות 1 וקיים קודקוד מטרה אחד בעומק d.
נניח שלפונקציה היוריסטית יש **שגיאה אבסולוטית קבועה**, כלומר: הפונקציה אינה underestimate לעלות האופטימלית להגיע למטרה ביותר מערך קבוע כלשהו: $h(n) = h^*(n) - k$.

<u>כמה קודקודים יורחבו לפי הנחה זו?</u>
נניח שהמסלול מהקודקוד ההתחלתי לקודקוד n מתפצל מהמסלול למטרה בקודקוד m. נניח שהמרחק מהקודקוד ההתחלתי למטרה הינו d, x הוא המרחק מהקודקוד ההתחלתי ל-m ו-y המרחק מ-m ל-n.



$h^*(n) = y + (d-x)$
$h(n) = y + d - x - k$
$f(n) = g(n) + h(n) = (x+y) + (y + d - x - k) = 2y + d - k \le d$

ולכן: $y \le \frac{k}{2}$.

לכן, לכל קודקוד m במסלול האופטימלי, מספר הקודקודים מתחתיו שאינם במסלול האופטימלי שעומקם מתחת ל-m אינו גבוה מ-$\frac{k}{2}$ הוא $(b-1)b^{\frac{k}{2}-1}$ (שכן יש b-1 הסתעפויות מתחת ל-m שאינן במסלול האוטפימלי ו-b קודקודים מתחת לכל הסתעפות כזאת). כמו כן, מספר הקודקודים m שבנתיב האופטימלי שניתן לסטות מהם הם d-1.

לכן, מספר הקודקודים שהעלות שלהם אינה עולה על d הוא $(d-1)(b-1)b^{\frac{k}{2}-1}$.
אם נוסיף את d הקודקודים שבמסלול האופטימלי נקבל שזה בדיוק קבוצת הקודקודים שמורחבת ע"י A* במקרה הגרוע ביותר, ולכן $O(d)$ (b, k הם קבועים).

לכן, סיבוכיות הזמן של A* עם היוריסטיקה בעלת שגיאה אבסולוטית קבועה הינו לינארי בעומק הפתרון.
צריך לשים לב שהסיבוכיות היא אקספוננציאלית בערך השגיאה k.

סיבוכיות מקום: כיוון שכל הקודקודים שנוצרו נשמרים ב-open/closed lists סיבוכיות המקום זהה לסיבוכיות הזמן.

פרק 4: Linear Space Heuristic Search

אלגוריתמים בעלי סיבוכיות זיכרון לינארית בעומק החיפוש המקסימלי.

Iterative Deepening A* (IDA*):

IDA* מיישם את עקרון ה-iterative deepening לסמלוץ A* תוך שימוש בזיכרון לינארי ולא אקספוננציאלי.

IDA* הוא ל-A* כמו ש-DFID הוא ל-BFS:
מוסיפים cost threshold.
בכל איטרציה מבצעים DFS ולכל קודקוד מחשבים את $f(n) = g(n) + h(n)$.
אם $f(n) \leq threshold$ מרחיבים את הקודקוד, אחרת מקצצים את כל ההסתעפות של הקודקוד.
אם מגיעים לקודקוד מטרה עם עלות קטנה/שווה מה-threshold – מחזירים אותו. אחרת, מתחילים איטרציה נוספת עם threshold גדול יותר. ה-threshold החדש הוא מינימום העלות בין כל הקודקודים שקוצצו באיטרציה הקודמת. ה-threshold הראשוני הינו ערך עלות קודקוד ההתחלה.

אם קיים פתרון עם עלות סופית, IDA* ימצא ויחזיר אותו.

כיוון שהאלגוריתם הוא DFS וממומש ע"י פונקציה רקורסיבית, כאשר הפתרון נמצא המסלול נמצא במחסנית הרקורסיה. לפיכך, אין צורך לשמור במפורש מצבים קודמים או מצביעים לאבא.

משפט: בגרף בו לכל הקודקודים יש משקל חיובי מינימלי, ערכי היוריסטיקה לא שליליים ואינם overestimate לעלות האמיתית וקיים מסלול עם עלות סופית למטרה, אזי IDA* יחזיר מסלול אופטימלי למטרה [הוכחה ע"י אינדוקציה: בשלב I מניחים שיש קודקוד n שהינו במסלול האופטימלי. במהלך איטרציה I+1, קודקוד n יווצר שוב שכן ה-threshold גדול מזה שבאיטרציה I].

סיבוכיות זיכרון: סיבוכיות הזיכרון הינה העומק המקסימלי של מחסנית הרקורסיה. נניח שעלות הפתרון האופטימלית היא c ועלות הקשת המינימלית היא e. האורך המקסימלי של מסלול עם עלות קטנה/שוה ל-c הוא $\frac{c}{e}$, לכן עומק החיפוש המקסימלי הוא $\frac{c}{e} + 1$. כיוון ש-e הוא קבוע, סיבוכיות הזיכרון היא O(c).

סיבוכיות זמן:
במקרה הגרוע ביותר (הפונקציה אינה קונסיסטנטית) IDA* ירחיב באיטרציה האחרונה אותם קודקודים כמו A*.
עבור פונקציה קונסיסטנטית, באיטרציה האחרונה יורחבו כל הקודקודים שמקושרים לשורש בעלי עלות קטנה/שווה ל-c.
לפיכך, באיטרציות הקודמות, מספר הקודקודים שייווצרו כאשר ה-cost threshold הוא x:

$IDA(x) = \sum_{i=1}^{x} N(x)$. אם $N(x)$ גדל בצורה אקספוננציאלית ב-x אם branching factor של b, אזי $\frac{IDA(x)}{IDA(x-1)} = b$, כלומר: בכל איטרציה מספר הקודקודים שמפותח גדל אקספוננציאלית ב-b.

לכן, סיבוכיות הזמן של IDA* זהה לזו של A* (למרות שייתכן שהאלגוריתם עצמו ירוץ יותר מהר מ-A* במידה והפונקציה היוריסטית קונסיסטנטית, כיוון שהוא מבצע DFS).

רוב העבודה ב-IDA* נעשית באיטרציה האחרונה.

קל יותר לממש את IDA* מאשר את A* שכן הוא אלגוריתם DFS ולא צריך לזכור open/closed lists.

מגבלות של IDA*:

- כאשר לכל קודקוד יש עלות שונה, בכל איטרציה יפותח קודקוד אחד נוסף בלבד. סיבוכיות הזמן במקרה זה עבור A* היא $O(b^d)$ ואילו עבור IDA* היא $O(b^{2d})$ ($O(b^d)$ איטרציות שרובן מכילות $O(b^d)$ קודקודים).
- מרחב הבעיות עבור IDA* חייב להיות עץ:
  - אם ניתן להגיע לקודקוד ממספר מסלולים הוא יהיה מיוצג על-ידי מספר קודקודים שונים בעץ.
  - A* יכול להימנע משכפול קודקודים (שכן יש לו זיכרון), ואילו IDA* לא יכול (ב-DFS אין זיכרון) ← לכן, אם יש הרבה מעגלים קטנים בגרף ואין מגבלת זיכרון **כדאי להשתמש ב-A***.
  - IDA* אינו טוב עבור כל הבעיות. לדוגמה: בבעיית ה-TSP ידוע עומק הפתרון ולכן אין סיבה לחפש באיטרציות שונות בעומקים שונים.

סיכום:

DFID משתמש בפונקציית עלות שהיא העומק.

IDA* משתמש בפונקציית עלות שהיא $g(n)+h(n)$.

ניתן ליצור גרסאת iterative deepening עבור UCS כאשר פונקציית העלות היא $g(n)$.

Depth First Branch and Bound (DFBnB):

מניחים קיום של פונקציית עלות שניתן ליישם על פתרון חלקי, שמהווה חסם תחתון לעלות של הפתרון המשלים של פתרון חלקי זה לפתרון המלא.

מתי משתמשים בו?

- כאשר הפתרון האופטימלי נחוץ.
- בעץ אינסופי כאשר יש חסם עליון טוב לעומק הפתרון האופטימלי או לעלות הפתרון.
- טוב יותר מ-IDA* שכן IDA* מבזבז הרבה זמן כשיש הרבה עלויות שונות.

כיצד האלגוריתם עובד?

האלגוריתם דומה ל-DFS כאשר:

- מאתחלים את α לאינסוף.
- כשמוצאים את הפתרון הראשון (בלי הגבלת הכלליות, כשהענף השמאלי ביותר מסתיים) יש בידינו מועמד לפתרון ועלותו נשמרת ב-α.
- מנקודה זו ואילך, כל פעם שהעלות של המסלול החדש גדולה או שווה ל-α, הענף מקוצץ וממשיכים לבדוק את הקודקוד הבא.
- כל פעם שמגיעים למסלול עם עלות קטנה מ-α, מעדכנים את α ואת הפתרון האופטימלי.
- האלגוריתם מסתיים כשמסיימים לעבור על כל העץ.

כיצד ניתן לשפר את DFBnB?

1. **סידור קודקודים**: שימוש בסידור קודקודים כדי לקבל פתרון בעל עלות מינימלית כמה שיותר מהר. כתוצאה מהסידור נקבל קיצוץ רחב יותר של ההסתעפויות בהמשך החיפוש (לדוגמה: ב-TSP נסדר את הקודקודים בסדר עולה של המרחקים של ערים בנות והאבות שלהן).
2. **פונקציה היוריסטית**: נשתמש בפונקציה היוריסטית $f(n) = g(n) + h(n)$ להערכת כל קודקוד והשוואתו מול α כדי לשפר את האלגוריתם. $f(n)$ לא חייבת להיות מונוטונית לא עולה, אלא פשוט לא overestimate לעלויות האמיתיות.

איכות הפתרון: DFBnB מחזיר את הפתרון האופטימלי.

סיבוכיות הזיכרון: O(bd), שכן יש לייצר ולאחסן את כל הבנים של כל קודקוד במסלול הנוכחי = O(d) עבור כל קודקוד.

סיבוכיות זמן:
במקרה הטוב ביותר, הפתרון הראשון שנמצא הוא גם הפתרון האופטימלי ולכן $c = \alpha$ לכל אורך החיפוש. האלגוריתם עדיין חייב להרחיב את כל הקודקודים בעלי עלות קטנה מ-c ולכן סיבוכיות הזמן שלו תהיה זהה ל-A* עם אותה פונקציה היוריסטית.
כיוון ש-DFBnB הינו אלגוריתם DFS, אזי בגרף עם מעגלים הוא עשוי לעבור על כל הנתיבים השונים למצב מסוים, ובכך סיבוכיות הזמן שלו תהיה גדולה יותר מזו של A*, שכן A* יכול לזהות ולקצץ קודקודים כפולים.

להמשך ניתוח סיבוכיות הזמן נשתמש ב-abstract analysis model: נניח שיש עץ עם branching factor ועומק אחידים ולקשתות מוצמדות עלויות באופן רנדומי.
לדוגמה: אם לקשתות יש עלות של 0 או 1 בהסתברות 0.5, אם התוחלת של הבנים בעלי עלות 0 הינה גדולה מ-1 אזי DFBnB עם סידור קודקודים ירוץ בזמן פולינומי בעומק החיפוש. אם התוחלת קטנה מ-1 אזי DFBnB ירוץ בזמן אקספוננציאלי בעומק החיפוש (בדומה ל-BFS ו-UCS).

DFBnB הינו אלגוריתם מסוג **any time**. בכל רגע נתון ניתן לעצור אותו ולהחזיר פתרון (לאו-דווקא אופטימלי).

ID vs. DFBnB:

דומה:
- שניהם מבטיחים פתרונות אופטימליים בהינתן היוריסטיקות עם חסם תחתון.
- שניהם מבוססי DFS וסיבוכיות זיכרון לינארית.
- שניהם משתמשים בחסם עלות גלובלי.

שונה:
- ב-ID ה-cost threshold הוא תמיד **חסם תחתון** לפתרון האופטימלי והוא **גדל** במהלך האיטרציות, ואילו DFBnB מתחיל עם **חסם עליון** ומקטין אותו במהלך האיטרציות.
- שניהם מרחיבים **יותר** קודקודים מ-A* (עקב סיבוכיות המקום הלינארית), כאשר ID מרחיב קודקודים בעלי **עלות קטנה מ-*c*** אבל ייתכן שיבצע הרחבה זו כמה פעמים, ואילו DFBnB מרחיב קודקודים בעלי **עלות גדולה מ-*c***.

פונקציות עלות לא מונוטוניות:

$f(n) = g(n) + wh(n)$.

IDA* עם פונקציית עלות לא מונוטונית לא מחפש בעץ בסדר best first search.

Recursive Best First Search (RBFS):

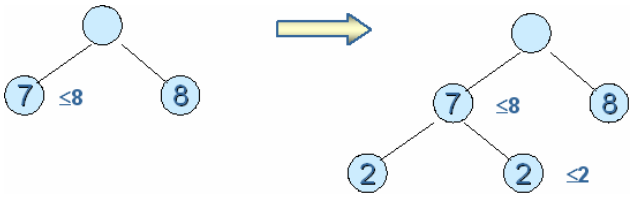
אלגוריתם עם סיבוכיות זיכרון לינארית.
האלגוריתם מרחיב קודקודים בסדר best first אפילו אם פונקציית העלות אינה מונוטונית. כמו כן, הוא מייצר **פחות** קודקודים מאשר ID כאשר פונקציית העלות היא כן מונוטונית.

Simple Recursive Best First Search (SRBFS):

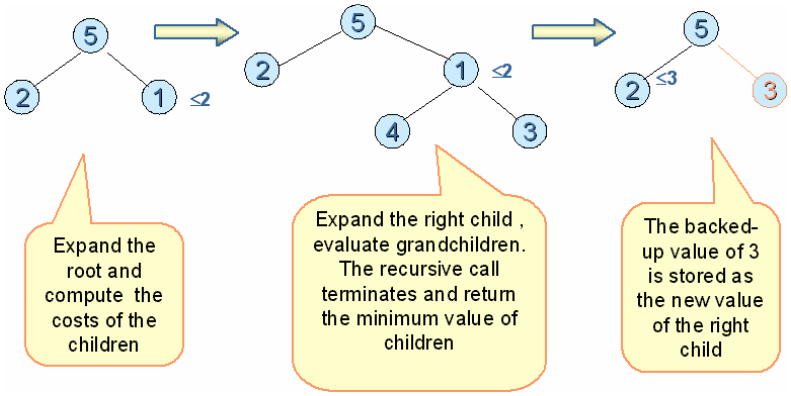
האלגוריתם משתמש ב-cost threshold **local** לכל קריאה רקורסיבית.
בכל קריאה מתקבלים 2 פרמטרים: קודקוד וחסם עליון.
האלגוריתם מבצע חיפוש בתת-עץ מתחת לקודקוד כל עוד יש בו קודקודים עם משקל קטן מהחסם העליון.

לכל קודקוד יש חסם עליון על העלות:
upper bound = min(upper bound of parent, current value of its lowest cost brother)
הקריאה הראשונה של האלגוריתם נעשית תמיד עם חסם עליון ∞.

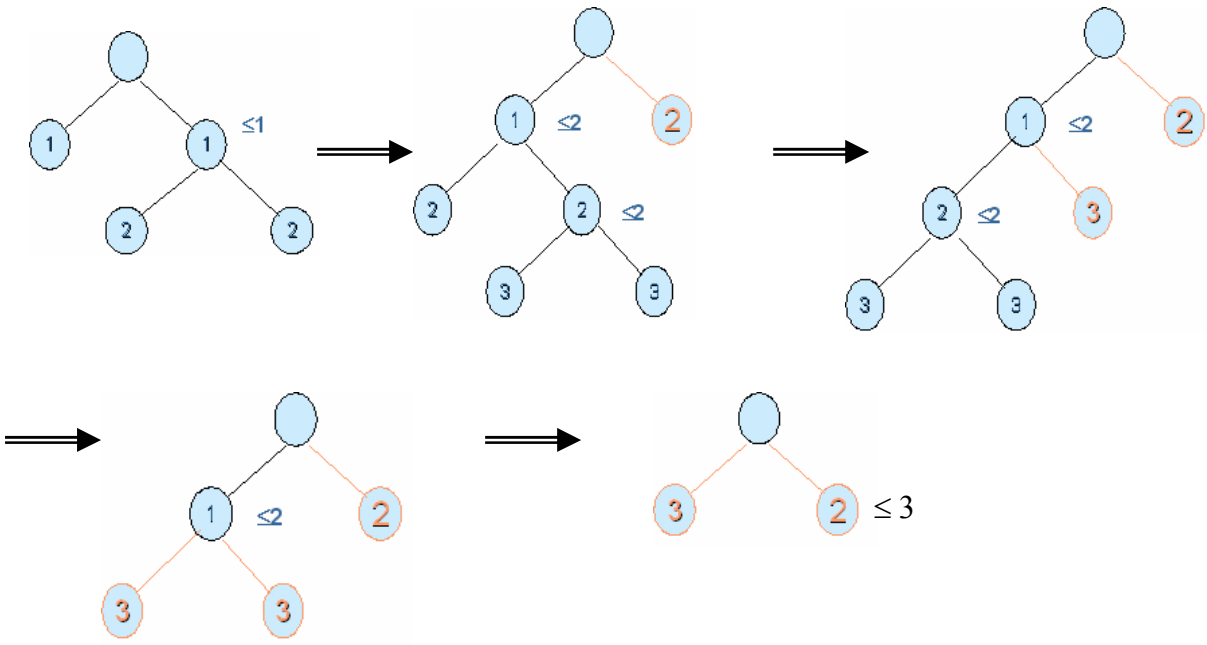
SRBFS **אינו יעיל** שכן הרבה מהעבודה שנעשית היא מיותרת (בכל חזרה מרקורסיה יורחבו שוב קודקודים עד הרמה בה נעצרו באיטרציות הקודמות). ניתן לשפר את זה אם הבנים יירשו את ערכי הוריהם אם הערכים האלו גדולים מהערכים שלהם.



דוגמה כאשר פונקציית העלות אינה מונוטונית:



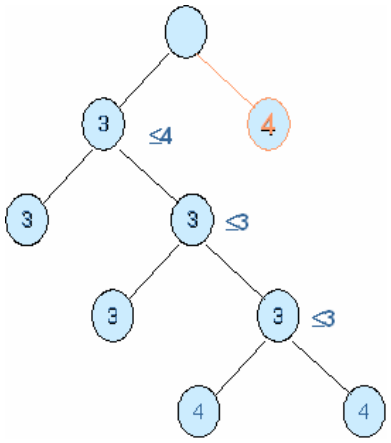
דוגמה כאשר יש משקלים אחידים:



Full Recursive Best First Search:
אם קודקוד לא הורחב בעבר אזי הערך המאוחסן שלו שווה לערך הסטטי (המקורי) שלו.

כדי להרחיב קודקוד החסם העליון חייב להיות לפחות גדול כמו הערך שאוחסן.
אם קודקוד הורחב בעבר אזי הערך המאוחסן שלו יהיה גדול יותר מהערך הסטטי שלו.
אם הערך שאוחסן לקודקוד גדול יותר מהערך הסטטי שלו, אזי הערך המאוחסן הינו המינימום בין הערך שאוחסן בעבר עבור ילדיו. לפיכך, הערך המאוחסן לקודקוד הוא חסם תחתון לערכים של ילדיו.
באופן כללי, הערך המאוחסן להורה משורשר לילדיו שיורשים אותו רק אם הוא גדול הן מהערך הסטטי של ההורה והערך הסטטי של הילד.

FRBFS מקבל 3 ארגומנטים: קודקוד n, הערך המאוחסן של הקודקוד $F(n)$ וחסם עליון B.

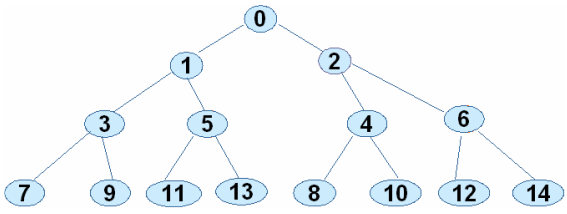


<u>התנהגות RBFS:</u>
עבור קודקודים חדשים ההתנהגות היא כמו best first search.
עבור קודקודים שהורחבו בעבר – מתנהג כמו DFS עד שמגיע ל-lowest cost node ואז מתנהג שוב כמו best first search.

סיבוכיות זיכרון של RBFS: O(bd). בכל זמן נתון מחסנית הרקורסיה מכילה את הנתיב ל-lowest cost frontier node בנוסף לכל האחים של הקודקודים בנתיב זה.

סיבוכיות זמן = מספר הקודקודים שהורחבו. תלוי בפונקציית העלות:
<u>במקרה הגרוע ביותר:</u>

- כאשר לכל הקודקודים יש עלות שונה.
- הקודקודים מסודרים כך שקודקוד בעל עלות שונה נמצא בתת-עץ שונה.



סיבוכיות הזמן דומה ל-ID: $O(b^{2d-1})$ (ב-ID הסיבוכיות היא $O(b^{2d})$). הסיבוכיות הזו נובעת מכך שיש $O(b^{d-1})$ קודקודים מהשורש ועד רמה d שצריך לפתח כדי למצוא עלות מינימלית. לפיכך, כל פיתוח של קודקוד חדש ברמה d דורש יצירה של $O(b^{d-1})$ קודקודים. כיון שיש b^d קודקודים בעומק d נדרש $O(b^{2d-1})$ זמן למציאת הפתרון.

<u>RBFS vs. ID:</u>

- אם פונקציית העלות אינה מונוטונית – לא ניתן להשוות בין האלגוריתמים.
- בממוצע, RBFS מפתח פחות קודקודים מאשר ID.

פרק 6: Design of Heuristic Functions

ניתן להגדיר היוריסטיקות ע"י ביצוע relaxation לבעיות (הורדת אילוצים ותנאים).

STRIPS – שפה עם פרדיקטים ואופרטורים.

1. קיימת רשימה של תנאי קדם.
2. add list – פרדיקטים שאינם "אמת" לפני ביצוע אופרטור ויהיו "אמת" לאחר הביצוע.
3. delete list – תת-קבוצה של תנאי הקדם שלאחר ביצוע האופרטור אינם נכונים יותר.

המצב מתואר ע"י קבוצת פרדיקטים שנכונים בנקודת זמן נתונה. רשימת תנאי הקדם מושווית לקבוצה זו כדי לקבוע אילו אופרטורים ניתנים להפעלה, כאשר הפעלת אופרטור מוסיפה/מוחקת פרדיקטים מרשימת הפרדיקטים התקפים.

היוריסטיקות שמתקבלות משיטה זו הן אדמיסביליות וקונסיסטנטיות (שקול להוספת מצבים וקשתות לגרף).

Pattern Database:

ניתן לחשב את הפונקציה היוריסטית ע"י טבלה, שלעתים יותר יעילה, שכן היא חוסכת זמן במהלך ריצת התוכנית.

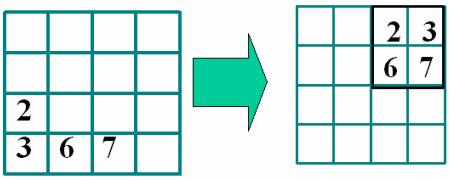
pattern database מאחסן את מספר הצעדים הנדרשים לפתור תבניות שונות של תתי הבעיה.

Pattern database זה בעצם lookup table ששומרת פתרונות לכל הקונפיגורציות של תת-בעיה/אבסטרקציה/תבנית של הבעיה הכללית. הטבלה משמשת כהיוריסטיקה במהלך החיפוש (לדוגמה: בקובייה ההונגרית 3x3x3 ניתן ליצור pattern databate עבור תת הבעיה של החלקים הקיצוניים 2x2x2).

קיימים סוגים שונים של pattern databases:

1. **non additive pattern databases:**

לדוגמה, להזזת חלקים 2,3,6,7 מהמיקום 8,12,13,14 בו הם נמצאים נתייעץ ב-P[8][12][13][14].



חסרון: ככל שהבעיה גדלה כך גם כמות הזיכרון שדורש ה-non additive pattern database גדלה בצורה מהירה מאוד.

אם יש כמה pattern databases ניקח את המקסימום שלהם.

לדוגמה, עבור מגדלי הנוי נוכל לחלק את הדיסקים לקבוצות זרות ונאחסן את העלות של מרחב התבנית הכללי לכל קבוצה ב-pattern database משלה. ערכי ה-pattern databases יבואו לידי ביטוי בהיוריסטיקה.

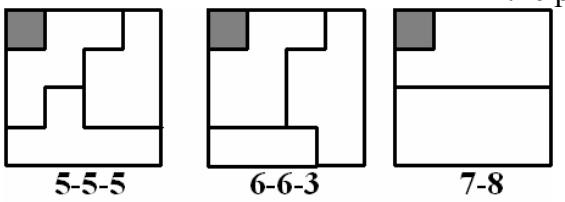
2. **disjoint additive pattern databases:**

ההבדל העיקרי בין non additive pattern databases ו-disjoint additive pattern databases הוא בעבודה ש-non additive pattern databases כוללים את כל המהלכים הנדרשים לפתרון הבעיה, כולל מהלכים שלא נכללים ב-pattern database עצמו. לכן, בהינתן 2 pattern databases, גם אם אין חפיפה ביניהם, אם נרצה ליצור פונקציה היוריסטית אדמיסבילית נהיה חייבים לקחת את ערך המקסימום של ה-databases ולא נוכל לבצע חיבור.

לעומת זאת, ערכים של pattern databases שהם **זרים** ניתן **לחבר** והם עדיין אדמיסבילים.

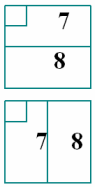
ניתן לבצע חיבוריות אם עלות תת-הבעיה מורכבת מעלויות של אובייקטים מתבניות מתאימות בלבד.

לדוגמה: עבור ה-tile puzzle:

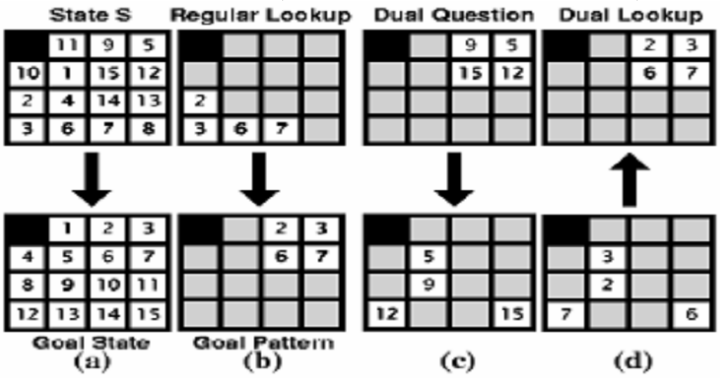


Manhattan Distances הוא דוגמה טריוויאלית ל-disjoint additive pattern database שבו כל database מכיל רק tile אחד.

- עדיף להשתמש בהרבה pattern databases קטנים ולקחת את המקסימום שלהם מאשר ב-pattern database אחד גדול.
- ישנה אפשרות לסימטריה ב-pattern database ולהשתמש בסימטריה זו לערך היורסיטי ע"י לקיחת ערך המקסימום מהחישובים הסימטריים. לדוגמה: בקובייה ההונגרית ניתן לבצע סיבוב של הקוביה. להלן דוגמה לשיקוף לאלכסון הראשי עבור ה-tile puzzle:



- לא חייבים לבצע רק סימטריה גיאוגרפית. ניתן להציג את הבעיה באופן דואלי ולהשתמש בנתונים אלו גם כן.
ההצגה הדואלית מתקבלת ע"י החלפה בין המשתנים לערכים.
בהצגה רגילה: משתנים = אובייקטים, ערכים = מיקום.
בהצגה דואלית: משתנים = מיקום, ערכים = אובייקטים.
לדוגמה: במקום להסתכל היכן נמצאים {2,3,6,7} וכמה צעדים צריך להזיז אותם ליעד נשאל מי נמצא במיקום {2,3,6,7} וכמה צעדים צריך להזיז אותם ליעד:



דוגמה נוספת:

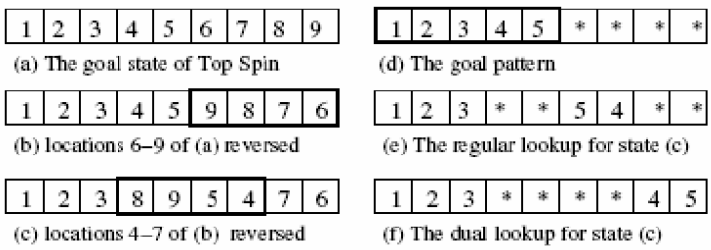


Figure 2: (9,4)-TopSpin states

- ניתן לבצע dual lookup כשיש סימטריה בין המיקומים והאובייקטים: כל אובייקט נמצא במיקום אחד וכל מיקום מכיל רק אובייקט אחד.
- צריך לשים לב ששימוש ב-dual lookup יכול להוביל ל-inconsistency.

פרק 7: 2 Player Perfect Information Games

שח:
1950 – Shannon: heuristic static evaluation function with minimax search.
1956 – McCarthy – α-β pruning.
1982 – Belle - פתרון חומרתי.
1997 – Deep Blue.

תוכנות ה-bridge והפוקר אינן טובות יותר מבני אדם.

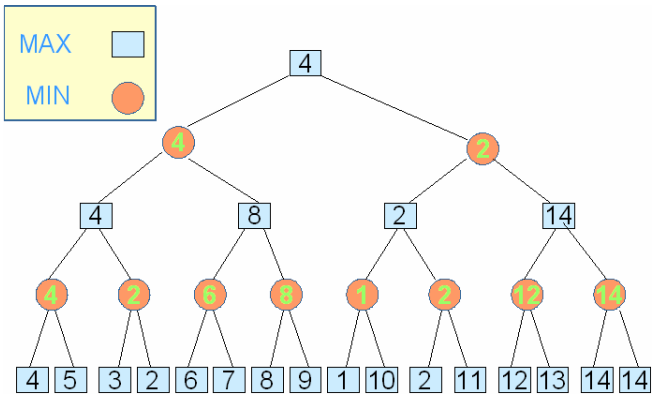
כלל אצבע: ככל שה-branching factor גדול יותר ← הביצועים של התוכנה גרועים יותר. יוצא מן הכלל: שש-בש.

Brute force search יעיל למשחקים קטנים בלבד (לדוגמה: 5 אבנים).

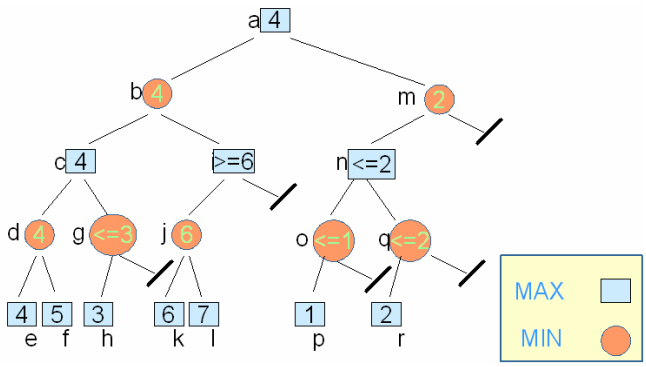
משפט ה-minimax: בכל משחק עם 2 שחקנים שהוא סכום אפס (zero sum) ניתן לכפות ניצחון או תיקו וניתן למצוא את אסטרטגיית ה-minimax.
לדוגמה: באיקס-עיגול – השורש יהיה תיקו.

כש-brute force search אינו ישים משתמשים ב-heuristic static evaluation function (=utility). בשח, לדוגמה, ה-utility יהיה ההפרש בין סכום הכלים כפול העוצמה של כל הכלים של שני השחקנים.

Minimax: מחפשים עמוק ככל שניתן בעץ, בהתחשב באילוצי הזמן והזיכרון.
כאשר תור MIN – שמור את המינימום של ערכי ילדיו.
כאשר תור MAX – שמור את המקסימום של ערכי ילדיו.
המהלך מתבצע לילד של השורש בעל הערך המקסימלי/מינימלי, בהתאם לתור השחקן (MIN או MAX).



α-β pruning:
מאפשר לחשב את ערך ה-minimax של השורש בלי לחשב את כל הקודקודים (דומה לרעיון של branch and bound).



היעילות של האלגוריתם תלויה בסדר בו מגיעים לקודקודים בחיפוש.

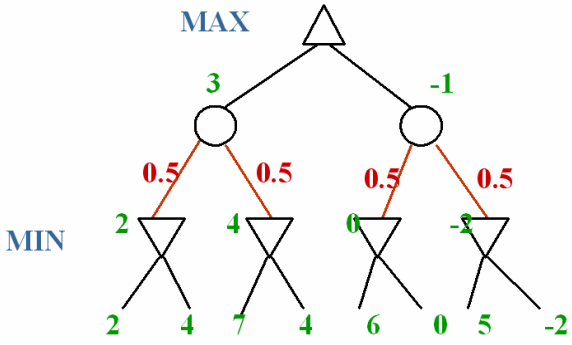
<u>במקרה הטוב ביותר</u>: אם הילד הגדול ביותר של קודקוד MAX מיוצר ראשון והילד הקטן ביותר של קודקוד MIN מיוצר ראשון (ניתן להגיע לסידור כזה ע"י **סידור קודקודים**) הסיבוכיות היא $O(b^{1/2})$.

<u>במקרה הגרוע ביותר</u>: $O(b)$. אם קודקודי ה-MAX נוצרים בסדר עולה ו-MIN בסדר יורד.

<u>במקרה הממוצע</u> – יש סדר רנדומי לקודקודים - $O(b^{3/4})$.

<u>משחקים עם מזל:</u>

chance nodes – קודקודים בהם מאורעות מזל יכולים לקרות (לדוגמה: הטלת קוביה/מטבע).
חישוב התוחלת נעשה ע"י ממוצע תוצאות ההסתברות.
לדוגמה:
c קודקוד מזל.
$P(d_i)$ – ההסתברות להטלת d_i.
$S(c,d_i)$ – קבוצת המיקומים שנוצרו ע"י יישום כל המהלכים החוקיים להטלת d_i בקודקוד c.



<u>שיפורים לאלגוריתם:</u>

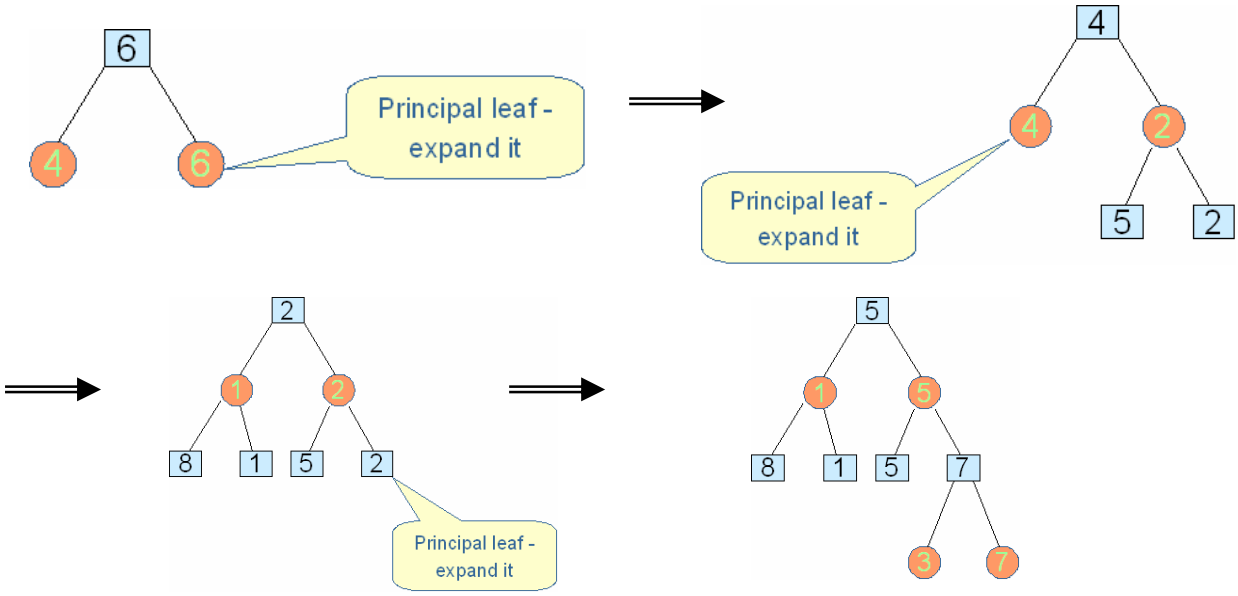
1. node ordering.
2. iterative deepening: כאשר הזמן נגמר מבצעים את המהלך בהתאם לאיטרציה האחרונה.
3. opening book: טבלה עם מהלכי "בית-ספר", בהתאם למומחיות אנשים.
4. end game database: DB עם מהלכי end game עם ערכי minimax.
5. special purpose hardware.
6. quiescence: ביצוע חיפוש משני כאשר הערכים אינם יציבים.
7. selective search: חיפוש רק של אזורים מעניינים ← best first minimax.
8. transposition table: שמירת טבלה עם מצבים שנוצרו בעבר יחד עם ערכיהם, כדי לזהות חזרה על מצבים. כשמגיע מצב והוא כבר קיים פשוט נלקח ערכו מהטבלה ולא מבוצע חיפוש נוסף.

Best First Minimax:

בהינתן עץ minimax פרוש חלקית, ערך ה-minimax שמשורשר לשורש נקבע לפי אחד מהעלים בעץ, כמו גם שאר הערכים במסלול מהשורש לעלה.

מסלול זה קרוי **principal variation** והעלה קרוי **principal leaf**. ה-principal variation הינו הניחוש הטוב ביותר של האלגוריתם בנוגע למהלכים האמיתיים שיבוצעו, בהינתן עץ המשחק החלקי. תמיד מרחיבים בתור הבא את ה-principal leaf, שכן הרציונל הוא שלקודקוד זה יש הכי הרבה השפעה להמלצה על המהלך הנוכחי.

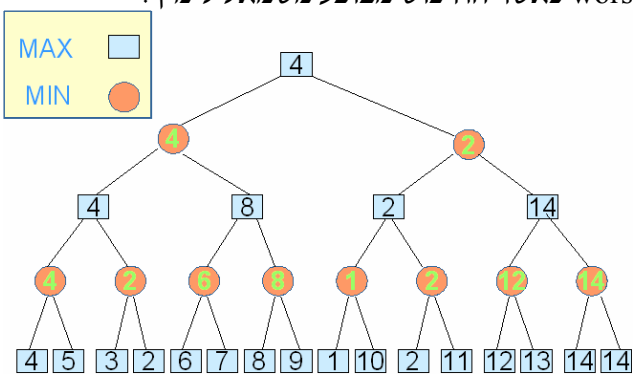
האלגוריתם ייצור unbalanced tree ויבצע החלטות שונות מאשר α-β full width fixed depth.



פרק 8: Performance Analysis of α-β Pruning

יעילות ה-α-β תלויה בסדר הקודקודים.

להלן דוגמה ל-worst case כאשר החיפוש מבוצע משמאל לימין:



ב-minimax מובטח שיוחזר ערך ה-minimax לשורש, ולכן כל אלגוריתם מסוג minimax חייב לבקר באסטרטגיית Max וחייב לבקר באסטרטגיית Min.

בהנחה שקיים branching factor אחיד ועומק אחיד, אזי באסטרטגיית Max נרחיב $b^{d/2}$ עלים.

כנ"ל עבור אסטרטגיית Min, ולכן סיבוכיות הזמן היא לפחות $O(b^{d/2})$.

פרק 9: Multi Player Games

משחקים לא שיתופיים בין מספר שחקנים. Perfect information.

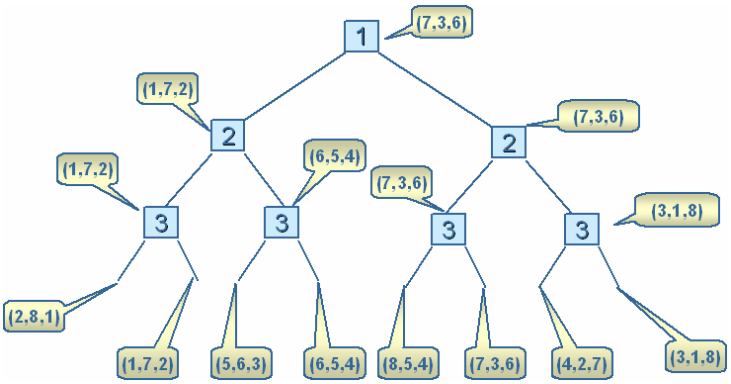
השחקנים משחקים לפי תורות, כל שחקן רוצה למקסם את התועלת שלו ואדיש בנוגע לפעילות השחקנים האחרים.

Max^n Algorithm:

פונקציית התועלת מחזירה וקטור עם n ערכים (ערך לכל שחקן). לדוגמה, באותלו הפונקצייה תחזיר את מספר החלקים של כל שחקן.

בכל קודקוד פנימי כשתור שחקן i לשחק, הערך המוחזר שווה לוקטור של הילד שעבורו ערך האלמנט ה-iי הינו מקסימלי.

דוגמה לעץ עבור שחקן $i = 3$:



<u>הגדרות:</u>

$M(x)$ – הערך היוריסטי הסטטי לקודקוד x.

$M(x,p)$ – הערך המשורשר של קודקוד x כשתור שחקן p בקודקוד x.

$M_i(x,p)$ – ערך $M(x,p)$ המתאים לתועלת של שחקן i.

$M(x_i,p') = \max M_p(x_i,p')$ ערך השחקן p' שמשחק אחרי p (הערך שווה למקסימום לכל הקודקודים הבנים של x).

$M(x,p) = M(x_i,p')$ או $M(x,p) = M(x)$ אם x הוא הקודקוד האחרון.

שבירת שוויון מתבצעת לטובת הקודקוד השמאלי ביותר.

- Minimax הוא סוג של Max^n כאשר n = 2 ופונקציית התועלת היא $(x, -x)$.
- בקודקודים בהם תור שחקן i, צריך לחשב רק את הרכיב ה-iי של הילדים, אולם זה לא יותר יקר לחשב את כל ערכי הרכיבים.
- אם אין שום הנחות בנוגע לערכי הרכיבים, ביצוע קיצוץ של ענפים הוא בלתי אפשרי (כשיש יותר מ-2 שחקנים).

<u>האלגוריתם הפרנואידי:</u>

באלגוריתם זה ההתייחסות היא לכל שאר השחקנים כאל **יריב אחד**.

אני משחק כ-MAX וכל שאר השחקנים הם MIN.

אופן הערכת העץ:

- כשתורי לשחק – חשב לפי **מקסימום** הערך שלי.
- כשלא תורי – חשב לפי **מינימום** הערך שלי.

- ב-Max^n מניחים שכל אחד מהשחקנים בוחר את המקסימום **שלו**, ואילו באלגוריתם הפרנואידי מניחים שכל אחד מהשחקנים האחרים בוחר את **המינימום שלי**.
- האלגוריתם הפרנואידי הוא בעצם Max^{n-1}Min.
- האלגוריתם הפרנואידי מכליל טוב יותר את minimax כיוון שהוא נותן איזשהו "ביטוח" לערך השורש.

- האלגוריתם הפרנואידי רץ מהר יותר מ-Max^n שכן הוא שקול לאלגוריתם minimax רגיל עם 2 שחקנים. בשלב ה-min ב-minimax אפשר לפרוש n-1 רמות בבת-אחת ללא צורך לחשב קודקודים פנימיים.
- האלגוריתם הפרנואידי ו-Max^n אינם בהכרח ניתנים להשוואה שכן הם דורשים הנחות שונות בנוגע לכוונות השחקנים:
  - Max^n מניח שכל שחקן רוצה להשיג את התועלת הגבוהה ביותר עבורו.
  - האלגוריתם הפרנואידי מניח שכל שאר השחקנים רוצים את רעתי.

פרק 12: Constraint Satisfaction Problems (CSPs)

ב-CSP מעניין אותנו הפתרון ולא דרך ההגעה, לפיכך גם המטרה מיוצגת בצורה לא מפורשת (דוגמאות: N-Queens, צביעת גרפים, סיפוק נוסחה בוליאנית).

ניתן לייצג בעיית CSP כקבוצת **משתנים**, קבוצת **ערכים** לכל משתנה וקבוצת **אילוצים** עבור ערכי המשתנים. לדוגמה, יכולים להיות מספר סוגים של אילוצים:
unary constraints: תת-קבוצה של כל הערכים האפשריים שניתן להשים למשתנה.
binary constraints: מציין אילו קומבינציות של השמות אפשריות עבור **זוג** ערכים.
ternary constraints: מגביל את קבוצת הערכים שניתן להשים ל-3 משתנים שונים בו-זמנית.

ייצוג דואלי:

ניתן לייצג בעיית CSP בצורה דואלית:
כל **משתנה** של הייצוג המקורי הופך ל**אילוץ** בייצוג הדואלי, וכל **אילוץ** בייצוג המקורי הופך ל**משתנה** בייצוג הדואלי. אם האילוצים בייצוג המקורי הם בינריים אזי ייצוג המשתנים בייצוג הדואלי יהיה באמצעות זוגות סדורים.
ההצגה הדואלית של הייצוג הדואלי זהה להצגה המקורית.
לאופן ייצוג הבעיה יש השפעה על יעילות פתרון הבעיה.

לדוגמה:
תשבץ 3x3 שכל מילה היא חוקית.
ייצוג 1: **משתנים**: **6** משתנים (לכל השורות והעמודות).
ערכים: כל מילה בת 3 תווים.
אילוצים: **9** אילוצים בינריים בין העמודות והשורות: מילה שנבחרה חייבת להיות בעלת אותה אות בריבוע המשותף.
ייצוג 2: **משתנים**: **9** משתנים (לכל ריבוע).
ערכים: 26 אותיות של האי"ב.
אילוצים: **6** אילוצים ternary: האותיות בכל שורה ועמודה חייבים ליצור מילה חוקית בת 3 אותיות.

Graph Coloring
ייצוג 1: **משתנים**: הקודקודים.
ערכים: הצבעים השונים.
אילוצים: כל זוג קודקודים סמוכים חייב להיות בצבע אחר.
ייצוג 2: **משתנים**: הקשתות.
ערכים: זוגות סדורים של צבעים.
אילוצים: קשתות שמחוברות לאותו קודקוד חייבות להציב אותו צבע לקודקוד.

פתרון בעיות CSP:

1. **brute force**:
מנסים את כל ההשמות האפשריות של הערכים למשתנים ובודקים אותם מול האילוצים. דוחים את ההשמות שמפרות את האילוצים.

2. **chronological backtracking**:
בוחרים סדר למשתנים ולערכים.
מבצעים השמה למשתנים באופן סדרתי – אחד אחרי השני.

ההשמה מבוצעת כך שכל האילוצים על כל המשתנים שכבר הושמו אינם מופרים.
אם למשתנה אין השמה חוקית אזי מבצעים backtracking למשתנה האחרון שהושם ומשימים בו את הערך הבא ברשימה.
האלגוריתם ממשיך עד שנמצאת השמה מלאה וקונסיסטנטית (הצלחה) או שמוצאים שכל ההשמות האפשריות מפירות את האילוצים (כישלון).
ניתן להמשיך את האלגוריתם גם לאחר הצלחה על-מנת למצוא את כל ההשמות האפשריות.

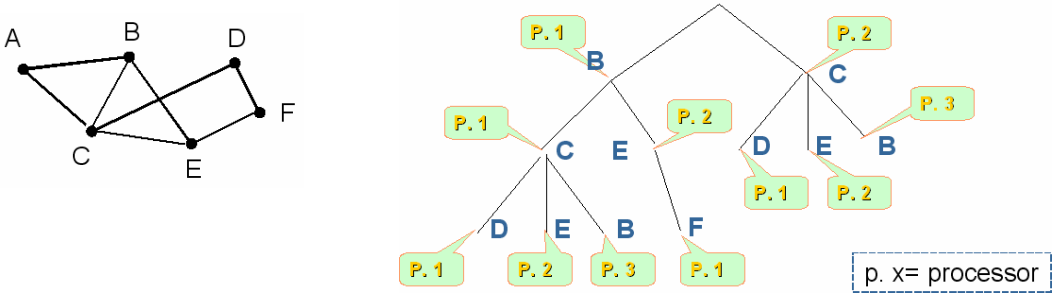
- האלגוריתם מאפשר לבצע קיצוץ של חלקים רחבים בעץ החיפוש.
- עומק העץ המקסימלי הוא כמספר המשתנים.
- ה-branching factor של כל קודקוד הוא כמספר הערכים שהוא יכול לקבל.
- כיוון שלעץ יש עומק קבוע וכל הפתרונות נמצאים בעומק המקסימלי ניתן להשתמש ב-DFS לחיפוש הפתרונות.
- ניתן לשפר את ה-backtracking ע"י:
  - **סידור משתנים**: סידור המשתנים מהמשתנה בעל הכי הרבה אילוצים לזה בעל הכי מעט אילוצים (ניתן לבצע את הסידור גם דינמית).
  - **סידור ערכים**: סדר הערכים קובע את הסדר בו יבוצע חיפוש בעץ. ניתן לסדר את הערכים מהכי כללי להכי מפורט על-מנת לצמצם את הזמן הדרוש למציאת הפתרון הראשון.
  - **back jumping**: כשמגיעים למבוי סתום, במקום לבצע undo להחלטה האחרונה מבצעים עדכון להחלטה שגרמה לכישלון.
  - **forward checking**: כשמבצעים השמה למשתנה, בודקים את כל המשתנים שטרם הושמו על-מנת לוודא שקיימת עבורם לפחות השמה אחת שהיא קונסיסטנטית. אם לא קיימת – מבצעים השמה אחרת למשתנה הנוכחי.
- בבעיות CSP חלק מהאילוצים אינם ניתנים בצורה מפורשת אלא נובעים מאילוצים אחרים, כאשר ניתן לגלות את האילוץ בצורה מפורשת רק תוך כדי החיפוש. בעיקרון, כשמגלים את האילוץ בצורה מפורשת עדיף לשמור אותו בצורה זו, על-מנת למנוע את הצורך בגילויו מחדש בחיפושים נוספים (ניתן ליישם ע"י arc consistency ו- path consistency).
- ניתן להסתכל על backtracking כעל סוג של branch and bound, כאשר קודקוד מקוצץ כאשר אילוץ מופר, פונקציית העלות שווה למספר האילוצים שהופרו עד כה, הפונקציה היא מונוטונית לא יורדת ועלות המטרה היא 0.

3. **תיקון היוריסטיקה**:
מחפשים במרחב הלא קונסיסטנטי שבו יש השמה מלאה לערכים עד אשר מוצאים השמה מלאה שהיא קונסיסטנטית.
לדוגמה: בבעיית N המלכות, כאשר כל המלכות על הלוח נבצע הזזות שלהן אחת-אחת עד שנמצא את הפתרון. המלכה הראשונה שמזיזים היא המלכה בעלת המספר הרב ביותר של התנגשויות.
חסרון: לא מובטח שיימצא הפתרון בזמן סופי. במידה שלא קיים פתרון, האלגוריתם ירוץ עד אינסוף, בעוד ששימוש ב-backtracking יוכל למצוא שהבעיה אינה פתירה.

פרק 13: Parallel Search Algorithms

1. **Parallel Node Generation**:
מקבול היצירה והערכת כל קודקוד.
מגבלות:
- השיטה הינה תלויית domain (יש לתכנן אלגוריתם בהתאם ל-domain).
- ניתן "לשאוב" מה-domain את מספר המעבדים הנחוץ אולם תוספת מעבדים מעבר לזה לא תגרום לזמן ריצה מהיר יותר.

לדוגמה:



2. **Parallel Window Search**:
הקצאת מעבדים שונים לניחושים שונים של המטרה. משתמשים בזה בעיקר ב-α-β וב-IDA*.

ב-α-β:
הרעיון: ככל ש-α ו-β הם יותר קרובים לערך התוצאה האמיתי, כך ניתן לקצץ יותר הסתעפויות.
בהינתן מספר מעבדים ניתן לחלק את הטווח $[-\infty,\infty]$ לתתי-קבוצות קטנות יותר ורציפות, ולתת לכל מעבד לחפש בכל העץ מהשורש עם ערכים התחלתיים שונים של α-β. אחד מהמעבדים יחזיר את ערך ה-min-max האמיתי ויבצע זאת בזמן מהיר יותר.
למרות זאת, הסיבוכיות נותרת $O(b^{d/2})$ (עדיין צריך לפתח את אסטרטגיית MAX ואסטרטגיית MIN).

ב-IDA*:
הרעיון: ככל שה-cutoff threshold קרוב יותר לתוצאה ניתן לפתח פחות קודקודים.
בהינתן מספר מעבדים ניתן להקצות לכל מעבד ערך cutoff שונה וכל מעבד יבצע חיפוש בכל העץ בהתאם לערך זה. כאשר המעבד מסיים את האיטרציה שלו הוא ממשיך לאיטרציה הבאה עם ערך cutoff שטרם הוקצה למעבדים השונים.
החיפוש המקבילי חוסך בעיקר את זמן בניית האיטרציה האחרונה שלפני ה-cutoff הנכון.
במקרה שאין פתרון, החיפוש המקבילי מפאשר רק speedup קטן יחסית בבניית כל האיטרציות.

3. **Tree Splitting Search**:
כל מעבד מחפש בחלקים שונים של העץ, כאשר ישנם 2 אלגוריתמים לחיפוש:

א. **Distributed Tree Search (DTS)**:
ב-DTS אין מגבלה על מספר המעבדים ואין דרישה לשליטה מרכזית במעבדים.
כל המעבדים מתחילים עם שורש העץ. אחד המעבדים מרחיב את השורש ויוצר את כל ילדיו. כל ילד מוקצה כעת למעבדים השונים.
כאשר אחד המעבדים מגיע לתוצאה בתת-העץ שלו הוא שולח הודעה למעבד שאחראי על קודקוד האבא. המעבד-האבא בודק האם זה הילד האחרון שטרם החזיר תוצאה. אם כן, המעבד מחשב את התוצאה ומחזיר אותה למעבד-אבא שלו. אחרת, המעבד שומר את התוצאה ומקצה את המעבד החופשי לקודקוד הילד הבא.

מגבלות:
- ניתן להשתמש רק כאשר התוצאות של תתי-העצים השונים אינן משפיעות אחת על השנייה.
- הקצאת מעבדים שגויה עשויה לפגוע במקביליות.

ב. **Branch and Bound Pruning**:
הקושי: באלגוריתם זה החיפוש בענף מסוים תלוי בתוצאה שמתקבלת מענף אחר.

האלגוריתם:
כמו ב-DTS מחלקים את p המעבדים באופן breadth first עד שלכל קודקוד יש מעבד.
כל מעבד מבצע חיפוש DFS מקבילי תוך שילוב התוצאות הקיימות.

מגבלות:
- מספר המעבדים יהיה בד"כ נמוך מאוד לעומת מספר העלים בעץ.
- לא ניתן לבצע מקבול לאלגוריתם עם יותר מ-bound אחד.

סיבוכיות ריצה:
נניח שה-branching factor שנוצר באלגוריתם הסדרתי הינו b^x, וזמן הריצה שלו הוא b^{xd}.
באלגוריתם המקבילי:
מבוצעת הקצאת המעבדים לפי breadth first בעומק $\log_b p$.
כל מעבד מבצע חיפוש בתת-העץ שלו שבעומק d - $\log_b p$. הזמן לפיכך הוא: $b^{x(d-\log_b p)}$.
שרשור התוצאה מכל המעבדים לוקח: $\log_b p + b^{x(d-\log_b p)}$, ולכן סה"כ זמן:
$b^{x(d-\log_b p)} + 2\log_b p \approx b^{x(d-\log_b p)}$.

סה"כ ה-speedup שמאפשר האלגוריתם המקבילי לעומת הסדרתי הוא: $\frac{b^{xd}}{b^{x(d-\log_b p)}} = p^x$.